[MAC0211] Laboratório de Programação I
Aula 7
Linguagem de Montagem
(Funções e Interfaces entre Programas em Linguagem de Montagem e C)

Kelly, adaptado por Gubi

DCC–IME–USP

15 de agosto de 2017

Implementação de funções

Considerações gerais:

- funções são implementadas como sub-rotinas (com CALL e RET)
- a passagem de parâmetros é feita via pilha
- a pilha também é usada para armazenar as variáveis locais da função
- o valor de retorno da função pode ser devolvido na pilha ou em EAX
- a função não deve "estragar" o valor dos registradores

Responsabilidade do chamador:

- empilhar parâmetros
- chamar função
- liberar espaço dos parâmetros

Implementação de funções

Responsabilidade da função chamada:

- salvar BP do chamador
- salvar todos os registradores que vão ser afetados
- alocar espaço para variáveis locais
- realizar trabalho usando argumentos e variáveis locais
- setar valor de retorno (em espaço próprio ou em EAX)
- desalocar o espaço das variáveis locais
- restaurar registradores afetados
- restaurar BP
- retornar

Implementação de funções

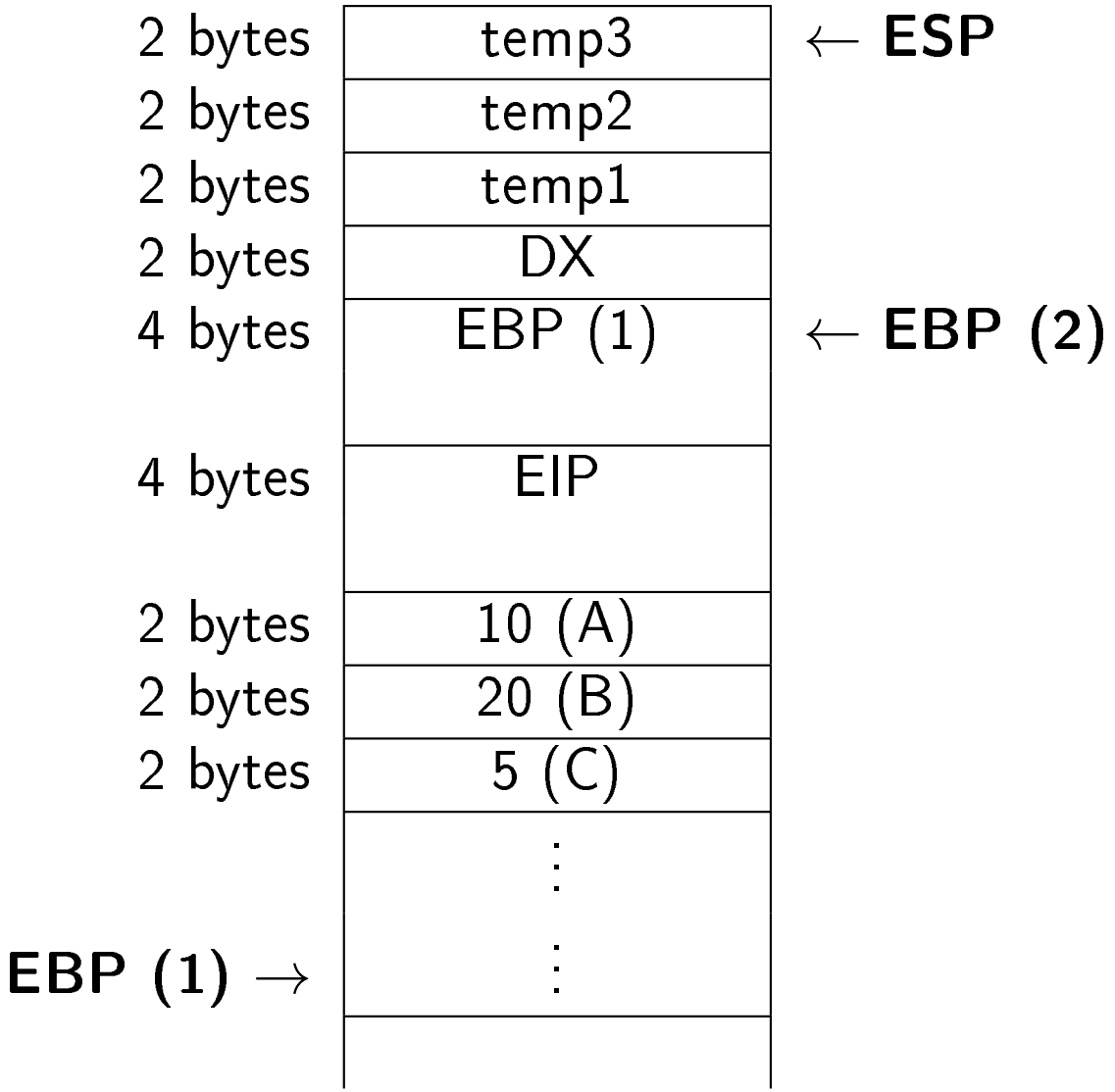
Exemplo – ver arquivos `funcao32.asm` e `funcao64.asm` no Paca

Implementação de uma função que possui o seguinte protótipo

```
int FUNC (int A, int B, int C)
```

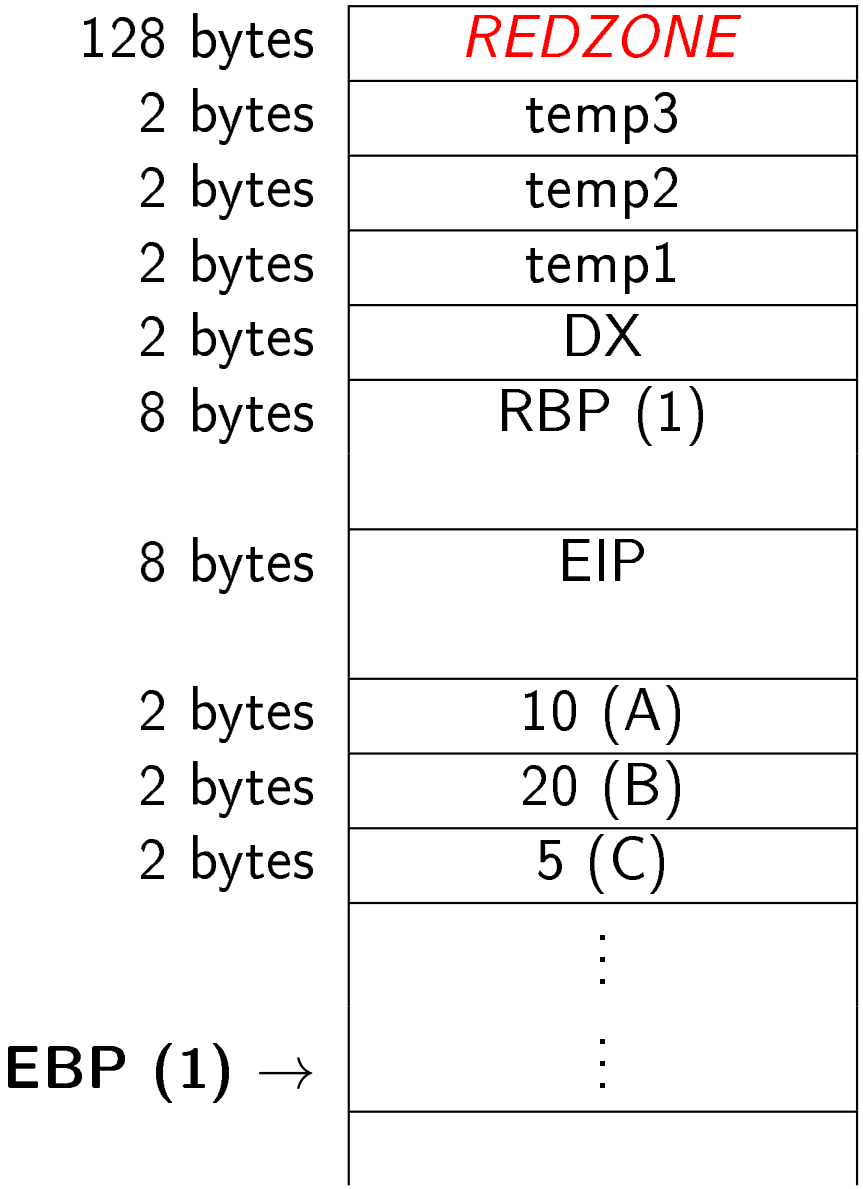
e que tem como saída o valor de $(A^2 + B^2)/C^2$.

Pilha de `funcao32.asm` [versão de 32 bits]



A	+8
B	+10
C	+12
temp1	-4
temp2	-6
temp3	-8

Pilha de `funcao64.asm` [versão de 64 bits]



A	+16
B	+18
C	+20
temp1	-4
temp2	-6
temp3	-8

Um "parênteses"...

Como gerar um executável para uma arquitetura de 32 bits usando um computador de 64 bits

Montagem:

```
$ nasm -f elf32 funcao32.asm
```

ou

```
$ as --32 -o funcao32.o funcao32.s
```

Ligação:

```
$ ld -m elf_i386 -o funcao32 funcao32.o
```

A execução de `funcao32` deve funcionar mesmo em computador de 64 bits.

Usando funções em linguagem de montagem a partir de programas em C

[Arquitetura de 32 bits] Convenções para funções

- parâmetros da função são passados pela pilha
- parâmetros são empilhados seguindo a ordem da direita para a esquerda (o último parâmetro é empilhado primeiro, o penúltimo é empilhado depois)
- valor de retorno da função é esperado em AL, AX ou EAX (depende do tipo do retorno)
- as convenções "tradicionais" para a implementação de funções também valem para C (ex.: definir stack frame, salvar registradores, alocar/desalocar na pilha espaço para variáveis locais, etc.)

Usando funções em linguagem de montagem a partir de programas em C

Exemplo: arquivos `prog.c` e `soma.c`

```c
#include <stdio.h>

int soma(int, int);

void main() {
  int x, y; x = 2; y = x++;
  printf("%d\n", soma(x,y++));
}
```

```c
int soma(int a, int b) {
  int x;
  x = a + b;
  return x;
}
```

Usando funções em linguagem de montagem a partir de programas em C

Exemplo [32 bits]: soma.c em linguagem de montagem

```
.global soma

var_x   = -4
param_a = 8
param_b = 12

soma:   push    %ebp                    # cria o stack frame
        mov     %esp, %ebp
        sub     $4,%esp                 # reserva espaco para x
        mov     param_a(%ebp), %eax     # obtem a
        add     param_b(%ebp), %eax     # soma a e b e armazena em eax
        mov     %eax,var_x(%ebp)        # armazena a soma em x
        mov     %eax,var_x(%ebp)        # valor de retorno esta em x
        add     $4,%esp                 # libera espaco de x
        pop     %ebp                    # restaura o apontador da base
        ret
```

Usando funções em linguagem de montagem a partir de programas em C

[Arquitetura de 64 bits] Convenções para funções

- os parâmetros da funçao são passados por meio dos registradores RDI, RSI, RDX, RCX, R8, R9 (nessa ordem). Se a função tiver mais do que 6 parâmetros, os demais são passados por meio da pilha
- parâmetros são passados na ordem da esquerda para a direita (o primeiro parâmetro fica em RDI, o segundo em RSI, etc.)
- os registradores mencionados acima, mais o RAX, R10 e o R11 são "estragados" na chamada da função. Por isso, podem ser usados na implementação da função sem serem salvos
- valores inteiros de retorno são passados em RAX (e RDX, se o valor de retorno tiver mais de 64 bits)
- O tipo `long` ocupa 64 bits, enquanto `int` ocupa 32. Na função `void foo(long a, int b)`, `a` é passado em RDI e `b` em ESI.

Usando funções em linguagem de montagem a partir de programas em C

Exemplo [64 bits]: soma.c em linguagem de montagem

```
.global soma

soma:
        push    %rbp                    # cria o stack frame
        mov     %rsp,%rbp
        sub     $4,%rsp                 # reserva espaco para x
        mov     %edi, (%rbp-$4)         # armazena o primeiro param. em x
        add     %esi, (%rbp-$4)         # soma os dois parametros de entrada
        mov     (%rbp-$4),%eax          # armazena x em eax (retorno da funcao)
        add     $4,%rsp                 # libera espaco de x
        pop     %rbp                    # restaura a base da pilha
        ret
```

Criando executáveis para programas "mistos" (C + linguagem de montagem)

Exemplo 1:

```
$ gcc -m32 -o prog prog.c soma.s
```

- -m32 : gera executável para arquit. de 32 bits (para 64, usar **-m64**)
- -o prog : usa "prog" como nome para o executável gerado

Observações:

- no gcc, a extensão do arquivo com o código em linguagem de montagem tem que ser `.s`
- para sintaxe Intel, incluir no `.s` diretiva "`.intel_syntax noprefix`"
- os comentários na sintaxe Intel precisam ser iniciados por "#"
- pode ser necessário instalar no Linux o pacote *gcc-multilib*

Criando executáveis para programas "mistos" (C + linguagem de montagem)

Exemplo 2: geração do código objeto + ligação

Montagem:

```
$ as --32 -o soma.o soma.s
$ gcc -c -m32 -o prog.o prog.c
```

Ligação:

```
$ gcc -o prog -m32 prog.o soma.o
```

Usando funções em C a partir de programas em linguagem de montagem [32 bits]

```
global main       ; no gcc, o rotulo de entrada padrao e' o main
extern printf

section .text

main:
     push    dword[num]     ; 2º param: um inteiro
     push    dword msg      ; 1º param: ponteiro para uma string
     call    printf         ; chamada a funcao printf
     add     esp,8          ; libera o espaco dos parametros
     ret

section .data

msg: DB 'Esse numero  -> %d <- deveria ser 1234.',10,0
num: DD 1234
```

Usando funções em C a partir de programas em linguagem de montagem [64 bits]

```
.global main
.extern printf

.text
main:
    mov     $msg,%rdi    # 1º param: ponteiro para uma string
    mov     num, %rsi    # 2º param: um inteiro
    mov     $0,%rax      # como printf e' uma funcao vararg,
                         # RAX deve conter a qtde de params
                         # nao inteiros para a funcao
    call    printf       # chamada a funcao printf
    ret

.data
msg: .string "Esse numero  -> %d <- deveria ser 1234.\n"
num: .word 1234
```

Gerando código em linguagem de montagem a partir de C

Exemplo:

```
$ gcc -S -m32 -masm=intel -fverbose-asm programa.c
```

- **-S** : gera código em linguagem de montagem
- **-m32** : gera código para arquitetura de 32 bits (para a de 64, usar **-m64**)
- **-masm=intel** : gera código na sintaxe da Intel (o padrão é a sintaxe da AT&T)
- **-fverbose-asm** : inclui no código comentários que (possivelmente) ajudam na sua compreensão

(Os códigos em linguagem de montagem gerados pelo `gcc` para os arquivos `prog.c` e `soma.c` estão no Paca)

Bibliografia e materiais recomendados

- Slides de uma aula da universidade de Princenton sobre funções em linguagem de montagem
  `http://www.cs.princeton.edu/courses/archive/spr11/cos217/lectures/15AssemblyFunctions.pdf`
- Convenções de chamada da arquitetura x86
  `http://en.wikipedia.org/wiki/X86_calling_conventions`
- Notas das aulas de MAC0211 de 2010, feitas pelo Prof. Kon
  `http://www.ime.usp.br/~kon/MAC211`

Cenas dos próximos capítulos...

- Nossa última aula sobre linguagem de montagem
  - Depuração de código
  - Acesso aos parâmetros passados via linha de comando
  - Mais exemplos de programas (recursão)